

Internationalization and Error Handling

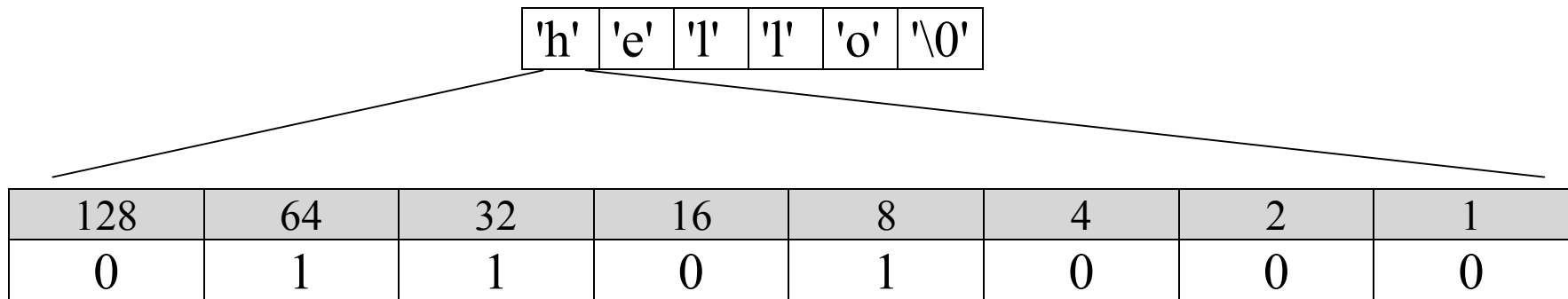
See also: Chapter 14 (397-400) and Chapter 15

Internationalization and Error Handling

CS193D, 3/6/06

Characters As We Know Them

Traditionally, programmers think of strings as an array of 8-bit characters:



(h is represented as #104)

8-bits (1 byte) can represent the numbers 0 through 255, so an 8-bit character can be used to represent up to 256 characters.

The ASCII Character Set

	Hex	0		1		2		3		4		5		6		7	
Hex.	Binary	0000		0001		0010		0011		0100		0101		0110		0111	
0	0000	NUL	0		16	SP	32	0	48	@	64	P	80	,	96	p	112
1	0001		1	DC1	17	!	33	1	49	A	65	Q	81	a	97	q	113
2	0010		2	DC2	18	ö	34	2	50	B	66	R	82	b	98	r	114
3	0011		3	DC3	19	#	35	3	51	C	67	S	83	c	99	s	115
4	0100		4	DC4	20	\$	36	4	52	D	68	T	84	d	100	t	116
5	0101		5		21	%	37	5	53	E	69	U	85	e	101	u	117
6	0110		6		22	&	38	6	54	F	70	V	86	f	102	v	118
7	0111	BEL	7		23	æ	39	7	55	G	71	W	87	g	103	w	119
8	1000	BS	8		24	(40	8	56	H	72	X	88	h	104	x	120
9	1001	HT	9		25)	41	9	57	I	73	Y	89	i	105	y	121
A	1010	LF	10		26	*	42	:	58	J	74	Z	90	j	106	z	122
B	1011	VT	11	ESC	27	+	43	;	59	K	75	[91	k	107	{	123
C	1100	FF	12		28	,	44	<	60	L	76	\	92	l	108		124
D	1101	CR	13		29	-	45	=	61	M	77]	93	m	109	}	125
E	1110	SO	14		30	.	46	>	62	N	78	^	94	n	110	~	126
F	1111	SI	15		31	/	47	?	63	O	79	_	95	o	111		127

The ASCII Character Set only goes up to 127, so it only uses 7 bits.

The Problem, Confounded

- No standardization of upper character assignments initially
- Lack of space meant per-region partitioning
- Per-region partitioning meant documents weren't interchangeable
- Everybody pretended that Asia wasn't there

Unicode

The Unicode standard maps characters to *code points*.

A maps to U+0041

The Klingon letter A (𐛀) maps to U+F8D0

Unicode has code points for many different languages, both modern and historic. You can see the Unicode charts at:

<http://www.unicode.org/>

There Are Still Some Issues

1. We haven't said anything about how a Unicode code point translates to an in-memory character.
2. We still have to deal with character sets that are larger than 256 characters.
3. If Unicode covers all languages, won't characters need to be huge?

Answer: Unicode is only half of the solution. We also need to know what *encoding* we are dealing with.

Two Popular Encodings

UTF-16: Uses 2 bytes (16 bits) for each character in the *Basic Multilingual Plane* and 4 bytes for other characters. Can represent any code point.

UTF-8: Uses 2 byte if possible. Somewhat backwards-compatible. Can also represent any code point.

In C++, multi-byte characters (a.k.a. *wide characters*) are represented by the type `wchar_t` and the size and encoding are platform dependent.

```
wchar_t myWideChar;
```

Using Multi-byte Characters in C++

```
wchar_t myWideChar = L'm';
```

```
wstring myWideString = L"This is a test.";
```

```
wofstream // wide char output file stream
```

```
wifstream // wide char input file stream
```

```
wcout
```

```
wcin
```

```
wcerr
```

Locales

Locales encompass the different representations of data in different regions.

```
wcout.imbue(locale("C"));  
wcout << 32767 << endl;  
wcout.imbue(locale("en_US"));  
wcout << 32767 << endl;  
wcout.imbue(locale("ru_RU "));  
wcout << 32767 << endl;
```

```
32767  
32,767  
32.767
```

Facets

A *facet* is a collection of information about a locale.

```
locale us("en_US");
locale gb("en_GB");

wstring c1, c2;

c1 = use_facet<moneypunct<wchar_t>>(us).curr_symbol();
c2 = use_facet<moneypunct<wchar_t>>(gb).curr_symbol();

wcout << c1 << endl;
wcout << c2 << endl;
```

\$

£

Error Handling

Review: Why are exceptions good?

- The throw/catch mechanism is distinct from return types.
- Lets the programmer decide whether to handle an error, or let the caller do it.
- Exceptions are objects and can contain data about the error.
- They can't be ignored (or else the program will terminate)
- They unwind the stack as they propagate up to the catch site.

About Throwing

- In C++, you can throw any type:

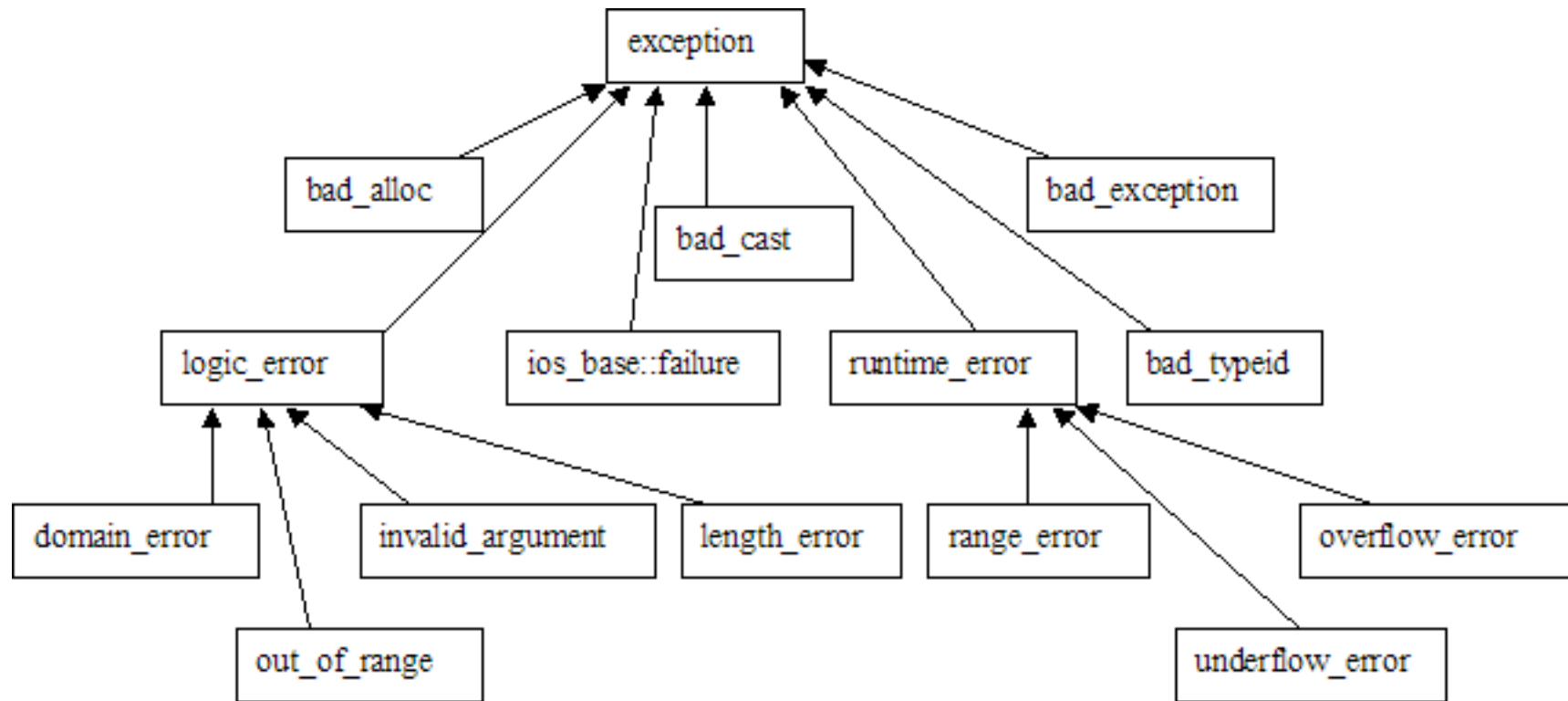
```
throw myInteger; // dubious style
throw "Error while reading file"; // possibly ok
throw exception(); // preferred
```

- A given piece of code can throw multiple exceptions:

```
if (i < 0) {
    throw "Attempted to use negative index";
} else if (i > maxIndex) {
    throw "Attempted to index past maxIndex";
}
```

- C++ has a built-in exception hierarchy that you can use and extend.
- You can describe the exceptions you might throw with a *throw list*.

The C++ Exception Hierarchy



Using Standard Exceptions

- Each class has a `what()` method that gives a human-readable description of the error.
- All constructors (except `exception()`) take the human-readable string as an argument.
- Hierarchy allows polymorphic catching of exceptions (more later).
- You can subclass the built-in exceptions to provide more specific exception types (example coming soon)

Throw Lists

In C++, throw lists are confusing, misleading, and borderline useless.

```
void myFunction() {    // no list - can throw anything
}
```

```
void myFunction()    // *should* only throw these two
    throw (invalid_argument, runtime_error) {
}
```

```
void myFunction()    // *shouldn't* throw any exceptions
    throw() {
}
```

If a function throws (or fails to catch) an exception that isn't in its throw list, C++ will call the `unexpected()` function, which halts the program by default.

Custom Unexpected Handlers

```
void readIntegerFile(string& inName, vector<int>& inVec)
    throw (runtime_error);
```

```
void myHandler() {
    cerr << "Unexpected exception!" << endl;
    throw runtime_error("");
}
```

```
int main() {
    vector<int> myInts;

    unexpected_handler old = set_unexpected(myHandler);
    try {
        readIntegerFile("IntegerFile.txt", myInts);
    } catch (const runtime_error & e) {
        [do something]
    }
    set_unexpected(old);
}
```

Subclassing the Standard Exceptions

Note: You can't change the throw list in an overridden method unless you make it *more* restrictive.

```
class FileError : public runtime_error {
public:
    FileError(const string& fname) :
        runtime_error(""), mFile(fname) {}
    virtual ~FileError() throw() {}

    virtual const char* what() const throw() {
        return mMsg.c_str();
    }

    string getFileName() { return mFile; }
protected:
    string mFile, mMsg;
};
```

```

class FReadError : public FileError
{
    public:
        FReadError(const string& fname, int line);
        virtual ~FReadError() throw() {}
        int getLineNum() { return mLineNum; }

    protected:
        int mLineNum;
};

FReadError::FReadError(const string& fname, int line) :
    FileError(fname), mLineNum(line) {
    ostringstream ostr;

    ostr << "Err reading " << fname << ", line " << line;

    mMsg = ostr.str();
}

```

Using the New Exceptions

```
void readIntFile(const string& fname, vector<int>& dest)
    throw (FileError)
{
    ifstream istr;
    int temp;
    char line[1024]; // reasonable assumption
    int lineNumber = 0;

    istr.open(filename.c_str());
    if (istr.fail()) {
        throw FileError(fname);
    }

    while (!istr.eof()) {
        istr.getline(line, 1024);
        lineNumber++;
        istringstream linestream(line);
        while (linestream >> temp) {
```

```
        dest.push_back(temp);
    }

    if (!linestream.eof()) {
        istr.close();
        throw FReadError(fname, lineNumber);
    }
}
istr.close();
}
```

```
// caller
try {
    readIntegerFile(filename, myInts);
} catch (const FileError& e) {
    cerr << e.what() << endl;
    exit(1);
}
```

Catching Exceptions

- Purpose: React to a situation where a "tried" block of code failed
- Matched by the type of the exception being caught.
- Can be caught by value, reference, or const reference (preferred):
`catch (const exception& e)`

- Multiple exceptions can be caught:

```
try {  
    doSomething();  
} catch (const SomeExceptionType& e) {  
    // handle SomeExceptionType  
} catch (const SomeOtherException& e) {  
    // handle SomeOtherExceptionType  
}
```

- Use ... as a catch-all:

```
    } catch (...) {  
        // do something  
    }
```
- The program terminates if nobody catches the exception.
- You can do something on an uncaught exception by setting a terminate handler, but there's not much you can do.
- Exceptions in a hierarchy can be caught polymorphically (e.g. handle all FileExceptions the same way). Be sure to catch by reference

Stack Unwinding

As control jumps from the exception throw site to the catch site, it skips any code but cleans up local objects and variables. Pointer values are *not* freed!

```
int main() {
    try {
        funcOne();
    } catch (const exception& e) {
    }
}
```

```
void funcOne() {
    string str1;
    string* str2 = new string();
    funcTwo();
    delete str2;
}
```

```
void funcTwo() {
    ifstream istr;
    istr.open("file");
    throw exception();
    istr.close();
}
```

The Old Catch 'n Rethrow

Aside from using smart pointers (or only using stack-based variables), which address the memory leak problem, you can catch exceptions, clean up, and rethrow them:

```
void funcOne() {
    string str1;
    string* str2 = new string();

    try {
        funcTwo();
    } catch (...) {
        delete str2;
        throw;    // rethrows the caught exception
    }
    delete str2;
}
```

There is no *finally* in C++!

Memory and Error Handling

By default, `new` and `new[]` will throw an exception if they can't allocate the desired memory.

```
try {
    ptr = new int[numInts];
} catch (bad_alloc& e) {
    cerr << "Unable to allocate memory! Gah!" << endl;
    return;
}
```

You can use C-style semantics with `nothrow-new`:

```
ptr = new(nothrow) int[numInts];
if (ptr == NULL) {
    cerr << "Unable to allocate! Save yourself!" << endl;
    return;
}
```

Handling Memory Errors with a New Handler

```
void myNewHandler()
{
    // we need to either free some memory, or terminate
    cerr << "Unable to allocate. Terminating." << endl;
    abort();
}

int main() {
    [etc]
    new_handler oldHandler = set_new_handler(myNewHandler);
    [etc]
}
```

Errors in Constructors and Destructors

- Exceptions are very useful to indicate that something went wrong in a ctor.
- If anything in a ctor can throw an exception, you should catch the exception, clean up allocated memory, and rethrow (next slide).
- If your superclass constructor throws an exception, C++ automatically destructs the subclasses (this is good).
- Destructors should not let exceptions be thrown. Another exception might already be unwinding, it's unclear what the caller would do, and you can leak memory.
- `delete` and `delete[]` do not throw exceptions.

```

GameBoard::GameBoard(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight) {
    int i, j;
    mCells = new GamePiece*[mWidth];

    try {
        for (int i = 0; i < mWidth; i++) {
            mCells[i] = new GamePiece[mHeight];
        }
    } catch (...) {
        // Clean up allocated memory
        for (j = 0; j < i; j++) {
            delete[] mCells[j];
        }
        delete[] mCells;

        throw;
    }
}

```